本科生实验报告

实验课程：操作系统原理实验

实验名称：编译内核/利用已有内核创建 OS

专业名称：计算机科学与技术

学生姓名：张玉瑶

学生学号：23336316

实验地点：实验楼 B203

实验成绩：

报告时间：2024 年 4 月 10 日

Section 1 实验概述

- 实验任务 1：
  1. 复现 Example 1，结合具体的代码说明 C 代码调用汇编函数的语法和汇编代码调用 C 函数的语法。
  2. 学习 make 的使用，并用 make 来构建 Example 1。
- 实验任务 2：完成...
- 实验任务 3：

  1. 在 src/8 的基础上，仿照 Example 3 编写段错误的中断处理函数，正确实现段错误的中断处理并正确地在中断描述符中注册。

  2.额外思考：使用尽可能多的方法触发段错误，并在实验报告里总结一下，引发段错误都有哪几种。
- 实验任务 4：复现 Example 4，仿照 Example 中使用 C 语言来实现时钟中断的例子，利用 C/C++、 InterruptManager、STDIO 和你自己封装的类来实现你的时钟中断处理过程，并通过这样的时钟中断，使用 C/C++语言来复刻 lab2 的 assignment 4 的字符回旋程序。

Section 2 实验步骤与实验结果

---------- 实验任务 1：混合编程的基本思路----------

- 任务要求：

  1.复现 Example 1，结合具体的代码说明 C 代码调用汇编函数的语法和汇编代码调用 C 函数的语法。

  2.学习 make 的使用，并用 make 来构建 Example 1。
- 思路分析：本实验通过 C++调用汇编函数，再由汇编调用 C 和 C++函数，展示混合编程机制。关键点包括：
  1. extern "C" 确保 C++函数名不被修饰，使汇编能正确调用。
  2. global 和 extern 在汇编中声明导出/导入函数。
  3. Makefile 自动化编译链接，处理不同语言的编译规则。
- 实验步骤：
  1. 在文件 c_func.c 中定义 C 函数 function_from_C。

```c
#include <stdio.h>

void function_from_C() {
    printf("This is a function from C.\n");
}
```

2. 在文件 cpp_func.cpp 中定义 C++函数 function_from_CPP。

```cpp
#include <iostream>


extern "C" void function_from_CPP() {
    std::cout << "This is a function from C++." << std::endl;
}
```

3. 在文件 asm_utils.asm 中定义汇编函数 function_from_asm，在 function_from_asm 中调用 function_from_C 和 function_from_CPP。

```
[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP

function_from_asm:
    call function_from_C
    call function_from_CPP
    ret
```

4.在文件 main.cpp 中调用汇编函数 function_from_asm。

```cpp
#include <iostream>

extern "C" void function_from_asm();

int main() {
    std::cout << "Call function from assembly." << std::endl;
    function_from_asm();
    std::cout << "Done." << std::endl;
}
```

5.c 代码调用汇编函数的语法：

1) main.cpp 调用 function_from_sam();

2) asm_utils.asm 调用 function_from_C()和 function_from_CPP()。

3) asm_utils.asm 的语法

```
global function_from_asm      ;声明 function_from_asm 为全局符号，可供其他模块调用
extern function_from_C        ;声明 function_from_C 为外部符号，将在其他模块中定义
extern function_from_CPP      ;声明 function_from_CPP 为外部符号，将在其他模块中定义

function_from_asm:
    call function_from_C      ;调用 C 函数 function_from_C()
    call function_from_CPP    ;调用 C++函数 function_from_CPP()
```

```
    ret                             ;返回
```

4）c_func.c 的语法

```
#include <stdio.h>

void function_from_C() {
    printf("This is a function from C.\n"); //打印引号中的内容
}
```

5）cpp_func.cpp 的语法

```
#include <iostream>

extern "C" void function_from_CPP() {//禁用 C++的名称修饰，使函数名保持为简单的"function_from_CPP"
    std::cout << "This is a function from C++." << std::endl; //打印引号中的内容
}
```

6）main.cpp 语法

```
#include <iostream>

extern "C" void function_from_asm(); //extern "C"用于声明汇编函数，确保名称匹配

int main() {
    std::cout << "Call function from assembly." << std::endl;
    function_from_asm();
    std::cout << "Done." << std::endl;
}
```

6.普通编译

```
zyy@VirtualBox:~/下载/sysu-2025-spring-operating-system-master/lab4/src/4$
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
g++ -o main.o -m32 -c main.cpp
nasm -o asm_utils.o -f elf32 asm_utils.asm
g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32
zyy@VirtualBox:~/下载/sysu-2025-spring-operating-system-master/lab4/src/4$
./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

7. 使用 make 来构建。

1）MakeFile 文件的编写。

```
main.out: main.o c_func.o cpp_func.o asm_utils.o
	g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o
  m32

c_func.o: c_func.c
	gcc -o c_func.o -m32 -c c_func.c

cpp_func.o: cpp_func.cpp
	g++ -o cpp_func.o -m32 -c cpp_func.cpp

main.o: main.cpp
	g++ -o main.o -m32 -c main.cpp

asm_func.o: asm_utils.asm
	nasm -o asm_utils.o -f elf32 asm_utils.asm
clean:
	rm *.o
```

2）编译运行，展示结果。

```
zyy@VirtualBox:~/下载/sysu-2025-spring-operating-system-master/lab4/src/4$ make
g++ -o main.o -m32 -c main.cpp
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
make: *** 没有规则可制作目标“asm_utils.o”，由“main.out” 需求。 停止。
zyy@VirtualBox:~/下载/sysu-2025-spring-operating-system-master/lab4/src/4$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

---------- 实验任务 2：使用 c/c++来编写内核----------

- 任务要求：复现 Example 2，在进入 setup_kernel 函数后，将输出 Hello World 改为输出你的学号。
- 思路分析：这个实验的核心思路是通过 bootloader 加载操作系统内核并完成控制权交接。主要分为以下几个步骤：

1．bootloader 完成保护模式初始化后，从硬盘第 6 扇区加载 200 个扇区的内核到内存 0x20000 地址处，然后跳转到该地址执行。

2．内核采用 C/C++和汇编混合编程，确保：内核入口代码(entry.asm)被放置在二进制文件开头，使用 objcopy 生成纯净的机器指令文件(kernel.bin)

3．目录结构采用标准 C/C++项目布局，通过 Makefile 管理编译过程，使用-I 参数简化头文件引用。

4．最终通过 dd 命令将 MBR、bootloader 和内核按顺序写入硬盘映像，实现完整的启动流程。

- 实验步骤：

1.文件目录结构。

```
├── build
│   └── makefile
├── include
│   ├── asm_utils.h
│   ├── boot.inc
│   ├── os_type.h
│   └── setup.h
├── run
│   ├── gdbinit
│   └── hd.img
└── src
    ├── boot
    │   ├── bootloader.asm
    │   ├── entry.asm
    │   └── mbr.asm
    ├── kernel
    │   └── setup.cpp
    └── utils
        └── asm_utils.asm
```

2.把读取内核的代码置于 bootloader 的最后。

```
mov eax, KERNEL_START_SECTOR
mov ebx, KERNEL_START_ADDRESS
mov ecx, KERNEL_SECTOR_COUNT

load_kernel:
    push eax
    push ebx
    call asm_read_hard_disk  ; 读取硬盘
    add esp, 8
    inc eax
    add ebx, 512
    loop load_kernel

jmp dword CODE_SELECTOR:KERNEL_START_ADDRESS ; 跳转到kernel
```

3. 常量的定义放置在 5/include/boot.inc 下，新增的内容如下。

```
; __________kernel_________
KERNEL_START_SECTOR equ 6
KERNEL_SECTOR_COUNT equ 200
KERNEL_START_ADDRESS equ 0x20000
```

4. 在 src/boot/entry.asm 下定义内核进入点。

```
global enter_kernel
extern setup_kernel
enter_kernel:
    jmp setup_kernel
```

5.将 setup_kernel 的定义在文件 src/kernel/setup.cpp 中。

```
#include "asm_utils.h"

extern "C" void setup_kernel()
{
    asm_hello_world();
    while(1) {

    }
}
```

6. 将汇编函数放置在 src/utils/asm_utils.h 下。

```
[bits 32]

global asm_hello_world

asm_hello_world:
    push eax
    xor eax, eax

    mov ah, 0x68 ;
    mov al, 'h'
    mov [gs:2 * 0], ax

    mov al, 'e'
    mov [gs:2 * 1], ax

    mov al, 'l'
    mov [gs:2 * 2], ax

    mov al, 'l'
    mov [gs:2 * 3], ax

    mov al, 'o'
    mov [gs:2 * 4], ax

    mov al, ' '
    mov [gs:2 * 5], ax

    mov al, 'w'
    mov [gs:2 * 6], ax

    mov al, 'o'
    mov [gs:2 * 7], ax

    mov al, 'r'
    mov [gs:2 * 8], ax

    mov al, 'l'
    mov [gs:2 * 9], ax

    mov al, 'd'
    mov [gs:2 * 10], ax

    pop eax
    ret
```

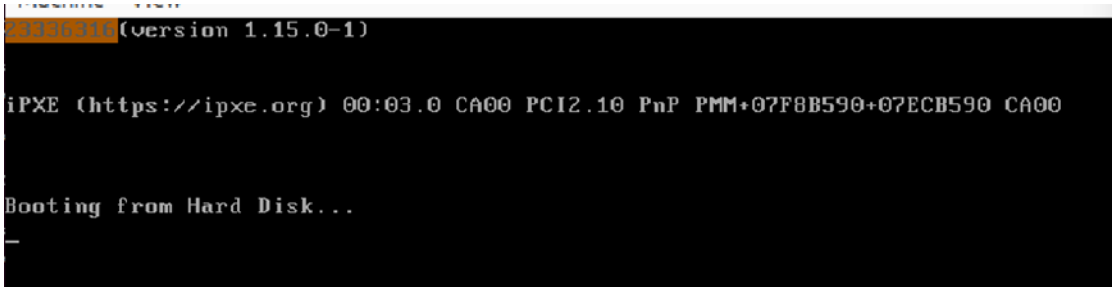
如果要输出 23336316，把代码改成如下。

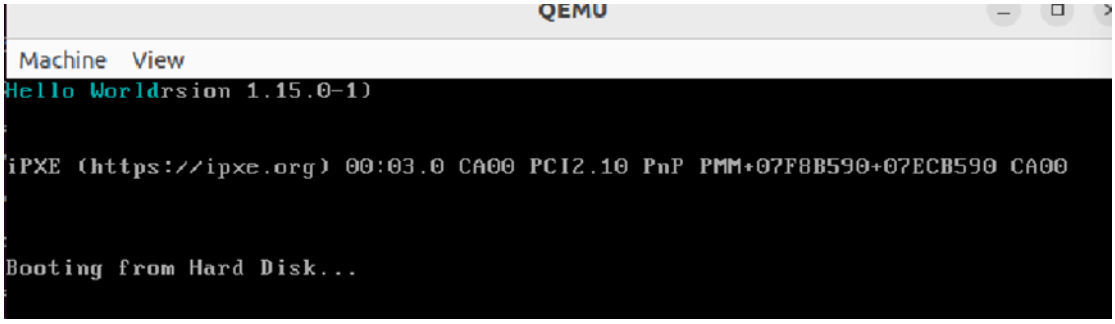
```
mov ah, 0x68 ;
mov al, '2'
mov [gs:2 * 0], ax

mov al, '3'
mov [gs:2 * 1], ax

mov al, '3'
mov [gs:2 * 2], ax

mov al, '3'
mov [gs:2 * 3], ax

mov al, '6'
mov [gs:2 * 4], ax

mov al, '3'
mov [gs:2 * 5], ax

mov al, '1'
mov [gs:2 * 6], ax

mov al, '6'
mov [gs:2 * 7], ax
```

7. 统一在文件 include/asm_utils.h 中声明所有的汇编函数，这样我们就不用单独地使用 extern 来声明了，只需要#include "asm_utils.h"即可。

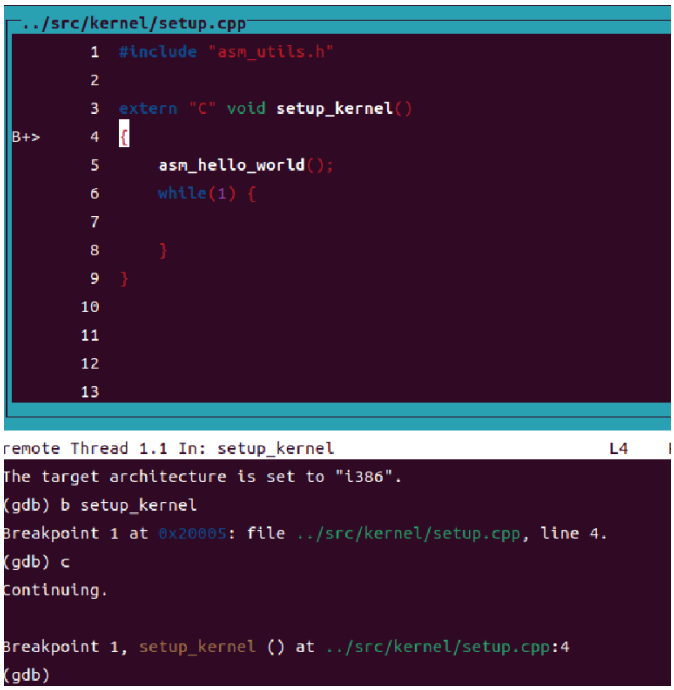
```
#ifndef ASM_UTILS_H
#define ASM_UTILS_H

extern "C" void asm_hello_world();

#endif
```

8.利用 make 进行编译。





9.利用 make debug 进行调试。



---------- 实验任务 3 ： 中断的处理----------

- 任务要求：

1. 在 src/8 的基础上，仿照 Example 3 编写段错误的中断处理函数，正确实现段错误的中断处理并正确地在中断描述符中注册。

2.额外思考：使用尽可能多的方法触发段错误，并在实验报告里总结一下，引发段错误都有哪几种。

- 思路分析：

本实验通过实现段错误的中断处理机制，深入理解操作系统的内存保护与中断管理。核心步骤包括：1) 定义中断管理器类（InterruptManager），初始化 IDT 并设置默认中断描述符；2) 编写段错误处理函数，通过 IDT 注册；3) 触发段错

误（如访问非法内存、空指针等），验证处理流程。实验重点在于掌握 IDT 结构、中断描述符配置及保护模式下的异常处理机制，同时通过多种方式触发段错误（如越界访问、权限违规），分析其原理与防护措施。

- 实验步骤：

1. 为了能够抽象地描述中断处理模块，定义一个类，称为中断管理器 InterruptManager，其定义放置在 include/interrupt.h 中。

```
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include "os_type.h"

class InterruptManager
{
private:
    uint32 *IDT;              // IDT起始地址

public:
    InterruptManager();
    void initialize();
    // 设置中断描述符
    // index    第index个描述符，index=0, 1, ..., 255
    // address 中断处理程序的起始地址
    // DPL      中断描述符的特权级
    void setInterruptDescriptor(uint32 index, uint32 address,
byte DPL);
};

#endif
```

2. 在 include/os_type.h 定义了基本的数据类型的别名。

```
#ifndef OS_TYPE_H
#define OS_TYPE_H

// 类型定义
typedef unsigned char byte;
typedef unsigned char uint8;

typedef unsigned short uint16;
typedef unsigned short word;

typedef unsigned int uint32;
typedef unsigned int uint;
typedef unsigned int dword;

#endif
```

3. 在使用中断之前，我们首先需要初始化 IDT。

1）初始化 IDT 的重要函数为 InterruptManager::initialize。

```
15 void InterruptManager::initialize()
16 {
17     // 初始化IDT
18     IDT = (uint32 *)IDT_START_ADDRESS;
19     asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);
20     for (uint i = 0; i < 256; ++i)
21     {
22         setInterruptDescriptor(i,
   (uint32)asm_unhandled_interrupt, 0);
23     }
```

2）将 IDT 设定在地址 0x8880 处，即 IDT_START_ADDRESS=0x8880。

```
4 #define IDT_START_ADDRESS 0x8880
```

3）在汇编代码中实现能够将 IDT 的信息放入到 IDTR 的函数 asm_lidt，代码放置在 src/utils/asm_utils.asm 中。将 IDT 的信息放入到 IDTR 后，我们就可以插入 256 个默认的中断处理描述符到 IDT 中。

```
; void asm_lidt(uint32 start, uint16 limit)
asm_lidt:
    push ebp
    mov ebp, esp
    push eax

    mov eax, [ebp + 4 * 3]
    mov [ASM_IDTR], ax
    mov eax, [ebp + 4 * 2]
    mov [ASM_IDTR + 2], eax
    lidt [ASM_IDTR]

    pop eax
    pop ebp
    ret
```

4.将段描述符的设置定义在函数 InterruptManager::setInterruptDescriptor 中。

```
27 void InterruptManager::setInterruptDescriptor(uint32 index,
   uint32 address, byte DPL)
28 {
29     IDT[index * 2] = (CODE_SELECTOR << 16) | (address & 0xffff);
30     IDT[index * 2 + 1] = (address & 0xffff0000) | (0x1 << 15) |
   (DPL << 13) | (0xe << 8);
```

5.编写段错误的中断处理函数。

1）定义默认的中断处理函数 asm_interrupt_empty_handler。函数首先关中断，然后输出提示字符串，最后做死循环。

```
; void asm_unhandled_interrupt()
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_INFO
    xor ebx, ebx
    mov ah, 0x03
```

```
.output_information:
    cmp byte[esi], 0
    je .end
    mov al, byte[esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    jmp .output_information
.end:
    jmp $
```

2）初始化页目录寄存器并启用分页机制的函数 asm_init_page_reg(int *directory)，将传入的页目录表地址（directory 参数）加载到 CR3 寄存器，设置 CR0 寄存器的 PG 位（第 31 位）为 1，启用分页机制。

```
; void asm_init_page_reg(int *directory);
asm_init_page_reg:
    push ebp
    mov ebp, esp

    push eax

    mov eax, [ebp + 4 * 2]
    mov cr3, eax ; 放入页目录表地址
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax          ; 置 PG=1，开启分页机制

    pop eax
    pop ebp

    ret
```

3）检查当前中断状态的函数 asm_interrupt_status()，可以查询当前 CPU 是否允许中断。先读取 EFLAGS 寄存器，然后提取中断标志位（第 9 位，0x200），最后返回该位的状态（0 表示中断禁用，非 0 表示中断启用）。

```
; int asm_interrupt_status();
asm_interrupt_status:
    xor eax, eax
    pushfd
    pop eax
    and eax, 0x200
    ret
```

4）禁用处理器中断函数 asm_disable_interrupt()，执行 cli 指令清除中断标志，在进入临界区前关闭中断。

```
; void asm_disable_interrupt();
```

```
asm_disable_interrupt:
    cli
    ret
; void asm_init_page_reg(int *directory);

asm_enable_interrupt:
    sti
    ret
```

5）启用处理器中断函数 asm_enable_interrupt()，执行 sti 指令设置中断标志，离开临界区后恢复中断。

```
asm_enable_interrupt:
    sti
    ret
```

6）asm_in_port(uint16 port, uint8 *value) 功能为从指定 I/O 端口读取一个字节。先使用 in 指令从端口读取数据，再将结果存储到传入的指针指向的内存位置，实现与硬件设备通信。

```
; void asm_in_port(uint16 port, uint8 *value)
asm_in_port:
    push ebp
    mov ebp, esp

    push edx
    push eax
    push ebx

    xor eax, eax
    mov edx, [ebp + 4 * 2] ; port
    mov ebx, [ebp + 4 * 3] ; *value

    in al, dx
    mov [ebx], al

    pop ebx
    pop eax
    pop edx
    pop ebp
    ret
```

7）asm_out_port(uint16 port, uint8 value)功能为向指定 I/O 端口写入一个字节，可以向硬件设备发送命令或数据。

```
; void asm_out_port(uint16 port, uint8 value)
asm_out_port:
    push ebp
```

```
    mov ebp, esp

    push edx
    push eax

    mov edx, [ebp + 4 * 2] ; port
    mov eax, [ebp + 4 * 3] ; value
    out dx, al

    pop eax
    pop edx
    pop ebp
    ret
```

6. 在 InterruptManager::initialize 最后，调用 setInterruptDescriptor 放入 256 个默认的中断描述符，这 256 个默认的中断描述符对应的中断处理函数是 asm_unhandled_interrupt。

```
20      for (uint i = 0; i < 256; ++i)
21      {
22          setInterruptDescriptor(i,
    (uint32)asm_unhandled_interrupt, 0);
23      }
```

7. 最后，我们在函数 src/kernel/setup_kernel.cpp 中定义并初始化中断处理器、输出管理器和内存管理器。

```
extern "C" void setup_kernel()
{

    // 中断管理器
    interruptManager.initialize();

    // 输出管理器
    stdio.initialize();

    // 内存管理器
    memoryManager.openPageMechanism();

    // 除零错误
    // int t = 1 / 0;

    // 段错误触发
    *(int*)0x100000 = 1;

    asm_halt();
}
```

8.在 include/os_modules.h 声明实例。

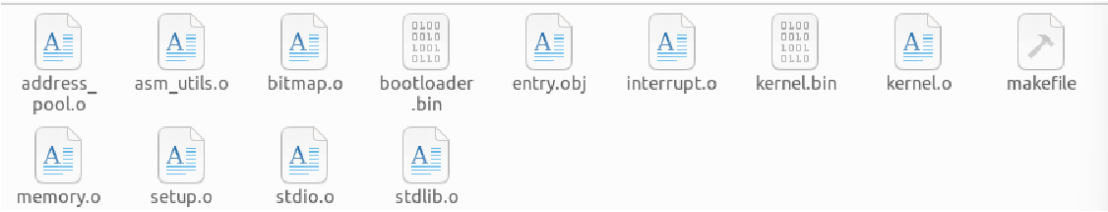
```
#ifndef OS_MODULES_H
#define OS_MODULES_H

#include "interrupt.h"
#include "stdio.h"
#include "memory.h"

extern InterruptManager interruptManager;
extern STDIO stdio;
extern MemoryManager memoryManager;

#endif
```

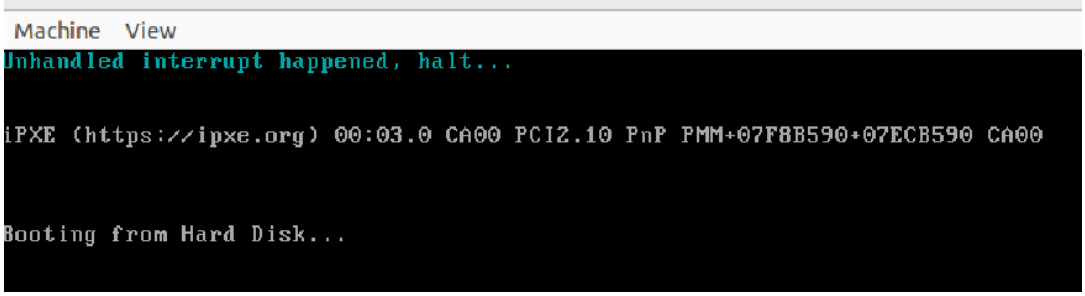
9. 将一些常量统一定义在文件 include/os_constant.h 下。

```
#ifndef OS_CONSTANT_H
#define OS_CONSTANT_H

#define IDT_START_ADDRESS 0x8880
#define CODE_SELECTOR 0x20
#define MAX_PROGRAM_NAME 16
#define MAX_PROGRAM_AMOUNT 16

#define MEMORY_SIZE_ADDRESS 0x7c00
#define PAGE_SIZE 4096
#define BITMAP_START_ADDRESS 0x10000

#define PAGE_DIRECTORY 0x100000

#endif
```

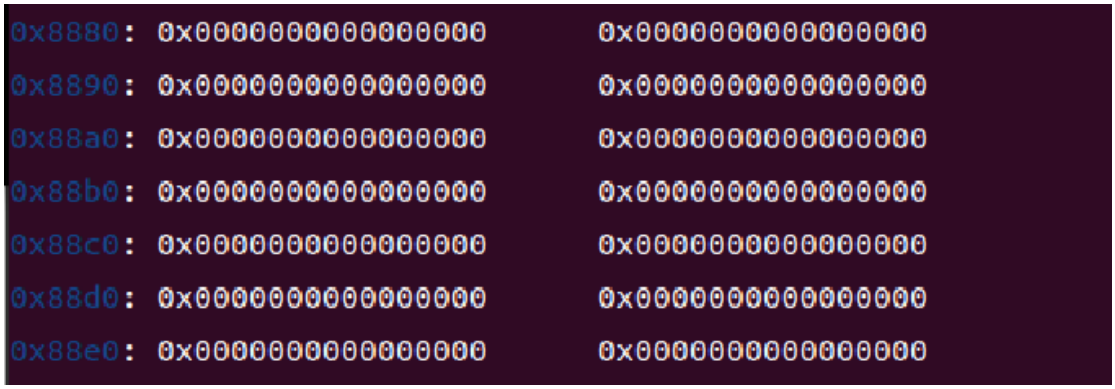
10.编译运行！中断被触发。

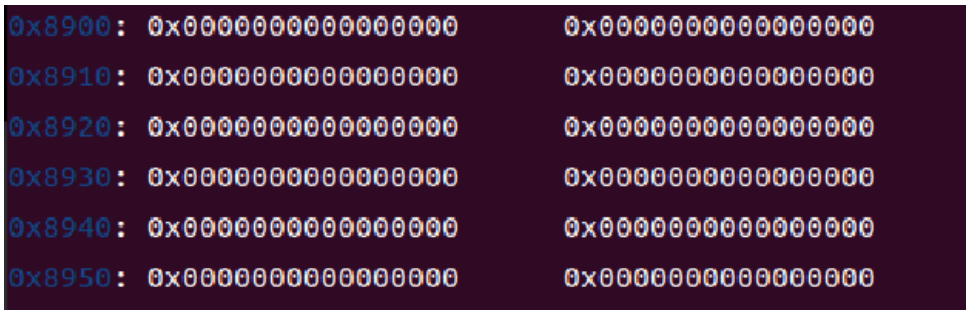
```
// 段错误触发
*(int*)0x100000 = 1;
```



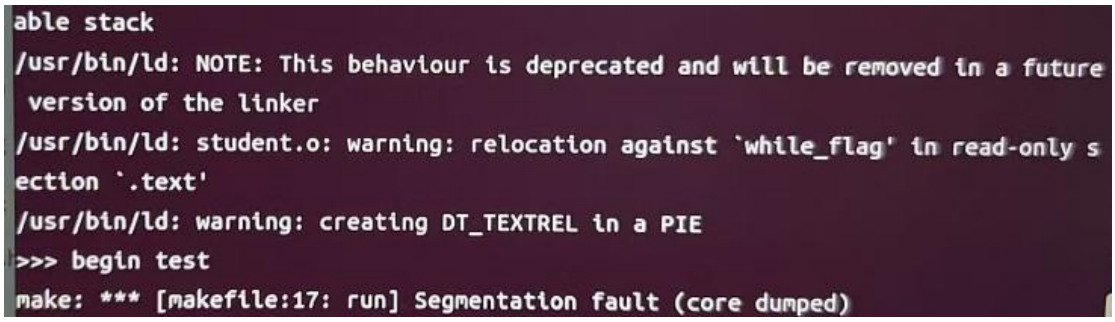


11.debug，在 gdb 下使用 x/256gx 0x8880 命令可以查看我们是否已经放入默认的中断描述符。





12.**其他段错误：**之前 lab2 的 assignment3 写的代码曾经触发过段错误，说明我的程序试图访问非法内存，导致操作系统强制终止程序并生成 core dump 文件。




```
your_while:
    mov ebx, [a2]
loop:
    cmp ebx, 25
    jge end_while
    call my_random
    shl ebx, 1
    mov ecx, [while_flag]
    add ebx,ecx
    mov [ebx], eax
```

```
    mov ecx,[a2]
    add ecx,1
    mov [a2],ecx
    mov ebx, [a2]
    jmp loop
end_while:
%include "end.include"
```

其他的段错误的触发方式：访问空指针，访问内核空间，栈溢出等。

思考题总结：因为实验内核通常采用平坦内存模型，未严格隔离内存区域，导致越界访问仍落在有效范围内，所以有时候“数组越界”等错误并不会触发中断。但是访问未映射内存、权限违规或执行不可执行代码会触发段错误，因为现代操作系统通过分页等机制来严格的检测错误，内核缺少这些保护，导致这些非法访问未被拦截威胁内核，因而显示报错。

---------- 实验任务 4 ----------

- 任务要求： 复现 Example 4，仿照 Example 中使用 C 语言来实现时钟中断的例子，利用 C/C++、 InterruptManager、STDIO 和你自己封装的类来实现你的时钟中断处理过程，并通过这样的时钟中断，使用 C/C++语言来复刻 lab2 的 assignment 4 的字符回旋程序。
- 思路分析：

本实验通过实现时钟中断处理机制，结合 C/C++和汇编完成动态字符显示功能。核心步骤包括：1) 初始化 8259A 中断控制器，配置时钟中断；2) 封装屏幕输出类（STDIO）管理光标和显示；3) 编写中断处理函数，实现实时时钟显示、固定学号输出及字符回旋动画。实验重点在于理解硬件中断机制、8253/8259A 芯片的编程方法，以及混合编程下的中断响应与处理流程，最终通过时钟中断驱动动态界面，复现字符回旋效果。

- 实验步骤：
  1. 复现 example4。按照教程不再赘述。
  2. 利用 C/C++、 InterruptManager、STDIO 和你自己封装的类来实现你的时钟中断处理过程，并通过这样的时钟中断，使用 C/C++语言来复刻 lab2 的 assignment 4 的字符回旋程序。

1） 为中断控制器 InterruptManager 加入如下成员变量和函数。

```
class InterruptManager
{
private:
    uint32 *IDT;              // IDT起始地址
    uint32 IRQ0_8259A_MASTER; // 主片中断起始向量号
    uint32 IRQ0_8259A_SLAVE;  // 从片中断起始向量号

public:
    InterruptManager();
    void initialize();
    // 设置中断描述符
    // index   第index个描述符，index=0, 1, ..., 255
    // address 中断处理程序的起始地址
    // DPL     中断描述符的特权级
    void setInterruptDescriptor(uint32 index, uint32 address,
byte DPL);
    // 开启时钟中断
    void enableTimeInterrupt();
    // 禁止时钟中断
    void disableTimeInterrupt();
    // 设置时钟中断处理函数
    void setTimeInterrupt(void *handler);

private:
    // 初始化8259A芯片
    void initialize8259A();
};
```

2)初始化 8259A 芯片，初始化的代码放置在成员函数 initialize8259A 中。

```
void InterruptManager::initialize8259A()
{
    // ICW 1
    asm_out_port(0x20, 0x11);
    asm_out_port(0xa0, 0x11);
    // ICW 2
    IRQ0_8259A_MASTER = 0x20;
    IRQ0_8259A_SLAVE = 0x28;
    asm_out_port(0x21, IRQ0_8259A_MASTER);
    asm_out_port(0xa1, IRQ0_8259A_SLAVE);
    // ICW 3
    asm_out_port(0x21, 4);
    asm_out_port(0xa1, 2);
    // ICW 4
    asm_out_port(0x21, 1);
    asm_out_port(0xa1, 1);

    // OCW 1 屏蔽主片所有中断，但主片的IRQ2需要开启
    asm_out_port(0x21, 0xfb);
    // OCW 1 屏蔽从片所有中断
    asm_out_port(0xa1, 0xff);
}
```

3）初始化 8259A 芯片的过程是通过设置一系列的 ICW 字来完成的。由于我们并未建立处理 8259A 中断的任何函数，因此在初始化的最后，我们需要屏蔽主片和从片的所有中断。其中，asm_out_port 是对 out 指令的封装，放在 asm_utils.asm 中。

```
; void asm_out_port(uint16 port, uint8 value)
asm_out_port:
    push ebp
    mov ebp, esp

    push edx
    push eax

    mov edx, [ebp + 4 * 2] ; port
    mov eax, [ebp + 4 * 3] ; value
    out dx, al

    pop eax
    pop edx
    pop ebp
    ret
```

4）接下来处理时钟中断，我们处理的时钟中断是主片的 IRQ0 中断。在计算机中，有一个称为 8253 的芯片，其能够以一定的频率来产生时钟中断。当其产生了时钟中断后，信号会被 8259A 截获，从而产生 IRQ0 中断。处理时钟中断并不需要了解 8253 芯片，只需要对 8259A 芯片产生的时钟中断进行处理即可。

I. 编写中断处理函数。我们希望能够像 printf 和 putchar 这样的函数来调用，因此，我们简单封装一个能够处理屏幕输出的类 STDIO。

```
#ifndef STDIO_H
#define STDIO_H

#include "os_type.h"

class STDIO
{
private:
    uint8 *screen;

public:
    STDIO();
    // 初始化函数
    void initialize();
    // 打印字符c，颜色color到位置(x,y)
    void print(uint x, uint y, uint8 c, uint8 color);
    // 打印字符c，颜色color到光标位置
    void print(uint8 c, uint8 color);
    // 打印字符c，颜色默认到光标位置
    void print(uint8 c);
    // 移动光标到一维位置
    void moveCursor(uint position);
    // 移动光标到二维位置
    void moveCursor(uint x, uint y);
    // 获取光标位置
    uint getCursor();

private:
    // 滚屏
    void rollUp();
};

#endif
```

接下来处理光标，屏幕的像素为 25*80，所以光标的位置从上到下，从左到右依次编号为 0-1999，用 16 位表示。与光标读写相关的端口为 0x3d4 和 0x3d5，在对光标读写之前，我们需要向端口 0x3d4 写入数据，表明我们操作的是光标的低 8 位还是高 8 位。写入 0x0e，表示操作的是高 8 位，写入 0x0f 表示操作的是低 8 位。如果我们需要需要读取光标，那么我们从 0x3d5 从读取数据；如果我们需要更改光标的位置，那么我们将光标的位置写入 0x3d5。

移动光标函数：

```
void STDIO::moveCursor(uint position)
{
    if (position >= 80 * 25)
    {
        return;
    }

    uint8 temp;

    // 处理高8位
    temp = (position >> 8) & 0xff;
    asm_out_port(0x3d4, 0x0e);
    asm_out_port(0x3d5, temp);

    // 处理低8位
    temp = position & 0xff;
    asm_out_port(0x3d4, 0x0f);
    asm_out_port(0x3d5, temp);
}
```

获取光标位置的函数：

```
uint STDIO::getCursor()
{
    uint pos;
    uint8 temp;

    pos = 0;
    temp = 0;
    // 处理高8位
    asm_out_port(0x3d4, 0x0e);
    asm_in_port(0x3d5, &temp);
    pos = ((uint)temp) << 8;

    // 处理低8位
    asm_out_port(0x3d4, 0x0f);
    asm_in_port(0x3d5, &temp);
    pos = pos | ((uint)temp);

    return pos;
}
```

封装 in 指令：

```
asm_in_port:
    push ebp
    mov ebp, esp

    push edx
    push eax
    push ebx

    xor eax, eax
    mov edx, [ebp + 4 * 2] ; port
    mov ebx, [ebp + 4 * 3] ; *value

    in al, dx
    mov [ebx], al

    pop ebx
    pop eax
    pop edx
    pop ebp
    ret
```

实现滚屏的函数：

```
void STDIO::rollUp()
{
    uint length;
    length = 25 * 80;
    for (uint i = 80; i < length; ++i)
    {
        screen[2 * (i - 80)] = screen[2 * i];
        screen[2 * (i - 80) + 1] = screen[2 * i + 1];
    }

    for (uint i = 24 * 80; i < length; ++i)
    {
        screen[2 * i] = ' ';
        screen[2 * i + 1] = 0x07;
    }
}
```

定义中断处理函数：c_time_interrupt_handler，每 18 次中断记作一秒，按时钟格式显示。实现功能：1.把计数器改成时钟格式显示；2。固定显示学号；3.实现字符串回旋。

Jump 类：实现移动、变色、变字符

```
struct Jump {
    int x = 0;
    int y = 0;
    int dir = 0;       // 0=右,1=下,2=左,3=上
    int charIdx = 0;
    int colorIdx = 0;

    const char* chars = "0123456789";
    const uint8_t colors[9] = {0x87,0x34,0x56,0x78,0x9a,0xab,0xec,
0x3e,0x4d};
};
```

I）. 把计数器改成时钟格式显示。

```
extern "C" void c_time_interrupt_handler()
{
    static Jump jump;
    ++ticks;

    // 每 18 次中断大约 1 秒
    if(ticks% 18 == 0) {
        ++seconds;

        char timeStr[9] = "00:00";
        uint32_t minutes = seconds / 60;
        uint32_t secs = seconds % 60;

        // 分钟十位和个位
        timeStr[0] = '0' + minutes / 10;
        timeStr[1] = '0' + minutes % 10;

        // 秒数十位和个位
        timeStr[3] = '0' + secs / 10;
        timeStr[4] = '0' + secs % 10;

        stdio.moveCursor(13,35);

        // 打印完整时间字符串
        for(int i = 0; i &lt; 5; ++i) {
            stdio.print(timeStr[i]);
        }
```

Ii）固定显示学号。

```
        if(!id_shown){


            const int center_row = 12;
            const int center_col = (80 - 8)/2;

            stdio.moveCursor(center_row, center_col);
            const char* student_id = "23336316";
            for(int i = 0; i &lt; 8; ++i) {
                stdio.print(student_id[i]);
            }

           id_shown = true;
        }
    }
```

Iii）每一次中断都更新一次回旋字符串。

```
        // 更新位置
        switch(jump.dir) {
            case 0:
                if(++jump.x &gt;= 24) {
                    jump.x=23;
                    jump.dir=1;
```

```
                } break;
            case 1:
                if(++jump.y &gt;= 79) {
                    jump.y=78;
                    jump.dir=2;
                  } break;
            case 2:
                if(--jump.x &lt; 0) {
                    jump.x=0;
                    jump.dir=3;
                } break;
            case 3:
                if(--jump.y &lt; 0) {
                    jump.y=0;
                    jump.dir=0;
                } break;
        }

        // 更新字符和颜色
        jump.charIdx = (jump.charIdx + 1) % 10;
        jump.colorIdx = (jump.colorIdx + 1) % 11;


        stdio.print(jump.x, jump.y,
                  jump.chars[jump.charIdx],
                  jump.colors[jump.colorIdx]);


    asm_out_port(0x20, 0x20); // EOI
}
```

完整的时钟处理函数。

```
asm_time_interrupt_handler:
    pushad

    ; 发送EOI消息，否则下一次中断不发生
    mov al, 0x20
    out 0x20, al
    out 0xa0, al

    call c_time_interrupt_handler

    popad
    iret
```

Ii. 设置时钟中断的中断描述符，也就是主片 IRQ0 中断对应的描述符。

```
void InterruptManager::setTimeInterrupt(void *handler)
{
    setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32)handler, 0);
}
```

封装一下开启和关闭时钟中断的函数。关于 8259A 上的中断开启情况，我们可以通过读取 OCW1 来得知；如果要修改 8259A 上的中断开启情况，我们就需要先读取再写入对应的 OCW1。

```
void InterruptManager::enableTimeInterrupt()
{
    uint8 value;
    // 读入主片OCW
    asm_in_port(0x21, &value);
    // 开启主片时钟中断，置0开启
    value = value & 0xfe;
    asm_out_port(0x21, value);
}

void InterruptManager::disableTimeInterrupt()
{
    uint8 value;
    asm_in_port(0x21, &value);
    // 关闭时钟中断，置1关闭
    value = value | 0x01;
    asm_out_port(0x21, value);
}
```

Iii & iv 在 setup_kernel 中定义 STDIO 的实例 stdio，最后初始化内核的组件，然后开启时钟中断和开中断。

```
extern "C" void setup_kernel()
{
    interruptManager.initialize();
    stdio.initialize();

    // 设置时钟中断处理程序
    interruptManager.setTimeInterrupt((void
*)asm_time_interrupt_handler);
    interruptManager.enableTimeInterrupt();

    asm_enable_interrupt();
    asm_halt();
}
```

声明实例。

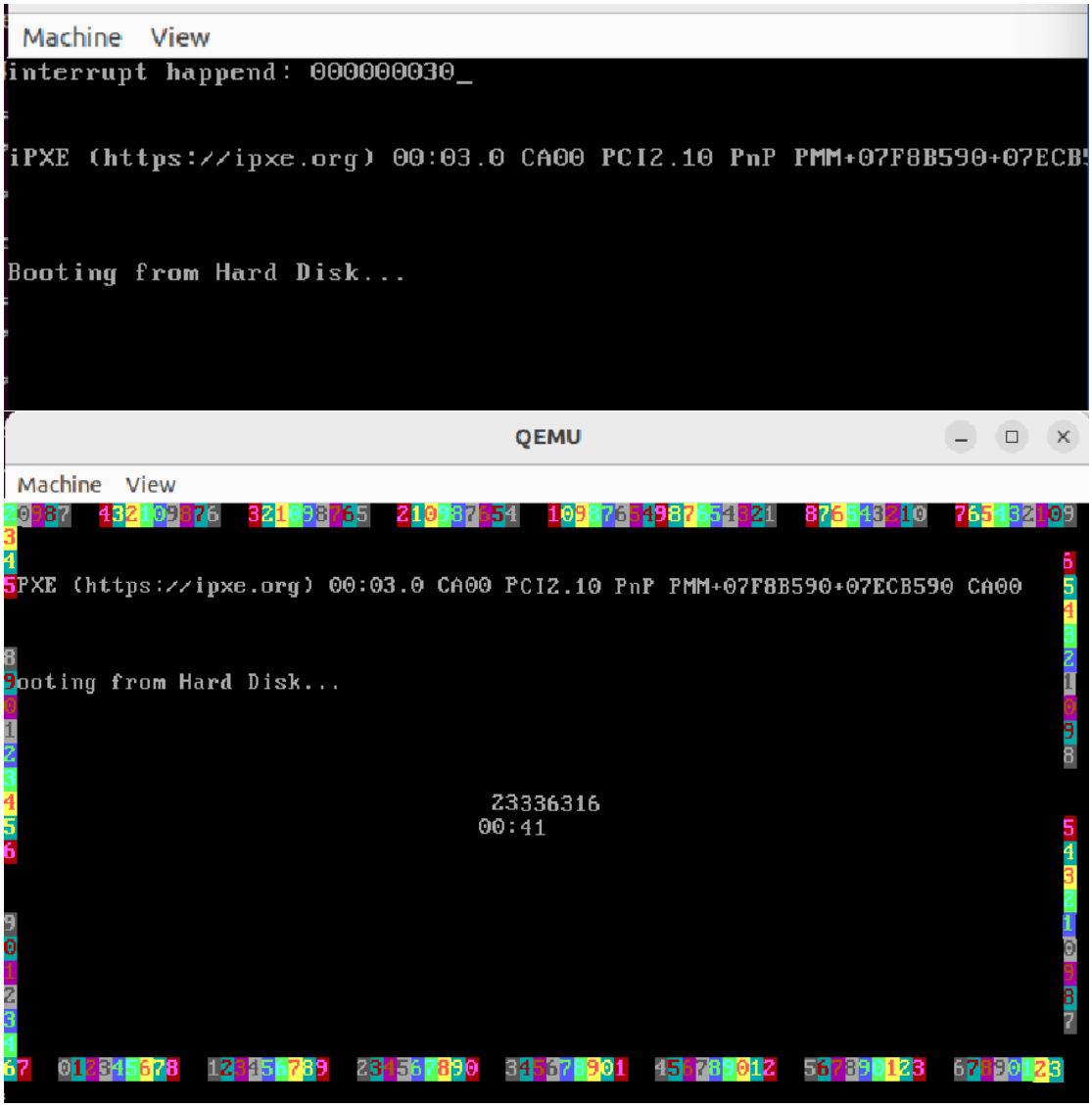
```
1 #ifndef OS_MODULES_H
2 #define OS_MODULES_H
3
4 #include "interrupt.h"
5 #include "stdio.h"
6
7 extern InterruptManager interruptManager;
8 extern STDIO stdio;
9
10 #endif
```

开中断指令。

```
asm_enable_interrupt:
    sti
    ret
```

最后编译运行。

- 实验结果展示：通过执行前述代码，可得下图结果。



Section 5 实验总结与心得体会

从这次实验中，我更深入理解了操作系统底层的中断处理机制，学习了混合编程，段错误的触发方式等。本次学习还是很有难度的，我会继续学习加深理解，为后续的实验和理论做好充分的准备。